



下载APP



13 | 用户登录：怎么判断线程中的Block原因？

2021-04-19 高楼

高楼的性能工程实战课

[进入课程 >](#)



讲述：高楼

时长 27:11 大小 24.91M



你好，我是高楼。

这节课我们接着来“玩”一下用户登录。在 [第 10 讲](#) 的课程中，我们以登录功能为例做了一些分析，来说明基准场景中的一些要点。但是，我们还没有把它优化完，所以这节课还要接着来折腾它。

用户登录说起来只是一个很普通的功能，不过它的逻辑一点也不简单。因为登录过程要对个人的信息进行对比验证，验证过程中又要调用相应的加密算法，而加密算法是对性能要求很高的一种功能。复杂的加密算法安全性高，但性能就差；不复杂的加密算法性能好，但安全性低，这是一个取舍的问题。

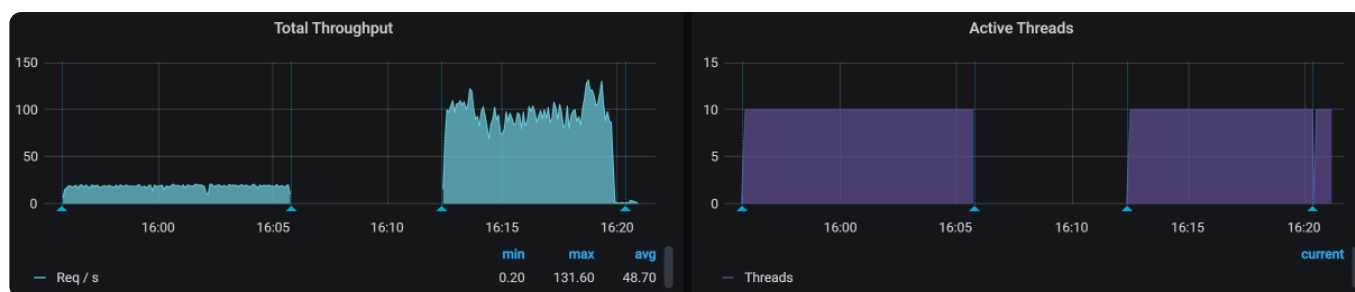


另外，还有 Session 存储和同步。对于个大型的系统来说，不管你在哪个系统访问，在调用其他系统时如果需要验证身份就要同步 Session 信息，并且在做业务时，我们也要把相应的 Session 信息带上，不然就识别不了。

你看，登录功能实际上会涉及到很多的业务，它其实一点也不简单。所以，这节课我会带着你好好分析用户登录功能，并带你了解在压力过程中业务逻辑链路和整体 TPS 之间的关系。同时，也希望你能学会判断线程中的 BLOCKED 原因。

修改加密算法

还记得在 [第 10 讲](#) 中，我们在基准场景中对登录业务的测试结果吗？在 10 个压力线程下，TPS 达到了 100 左右。

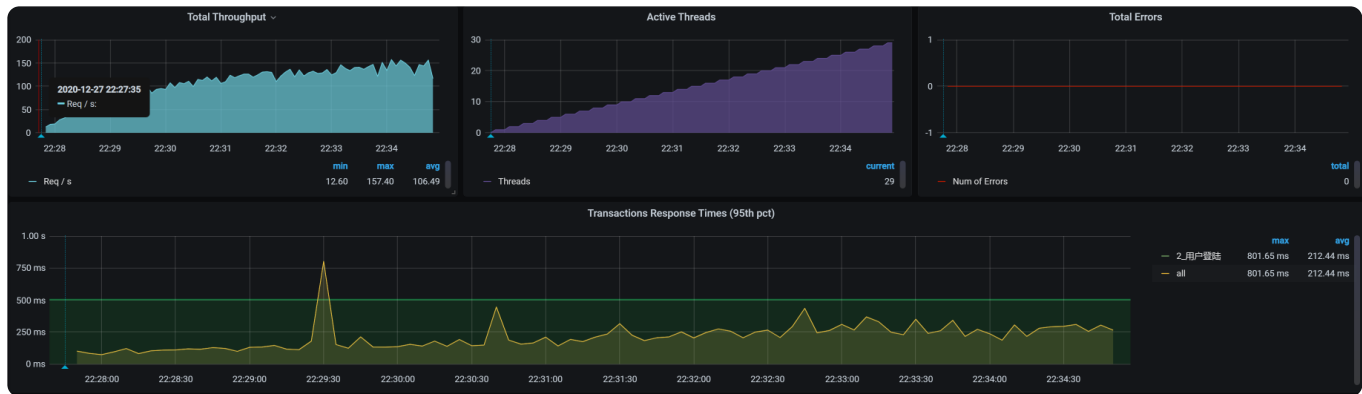


同时，在 [第 10 讲](#) 中，我们发现了加密算法 BCrypt 效率低之后，讨论了两种优化方式：一种是用更快的加密方式，另一种是去掉这个加密算法。当时，我选择把加密算法 BCrypt 直接去掉。在这节课中，我们来试试第一种方式，把它改为 MD5，具体有两个动作：

更改加密算法。之前的 BCrypt 加密算法虽然安全性高，但性能差，所以建议改成 MD5。

加载所有用户到 Redis 中。

我们再跑一遍压力场景。注意，在跑这一遍之前，我们只是更改了加密算法，并没有执行加载缓存的动作。我希望一次只做一个动作来判断结果（但是上面两个动作我们都要做哦，请你接着看下去），结果如下：



从上面的结果来看，性能有些上升了，但是还没达到我想要的样子。我希望性能有突飞猛进的增加，而不是现在这样不温不火的样子，看着就来气。所以，我们还是要继续“收拾收拾”这个接口，使用缓存，看下效果如何。

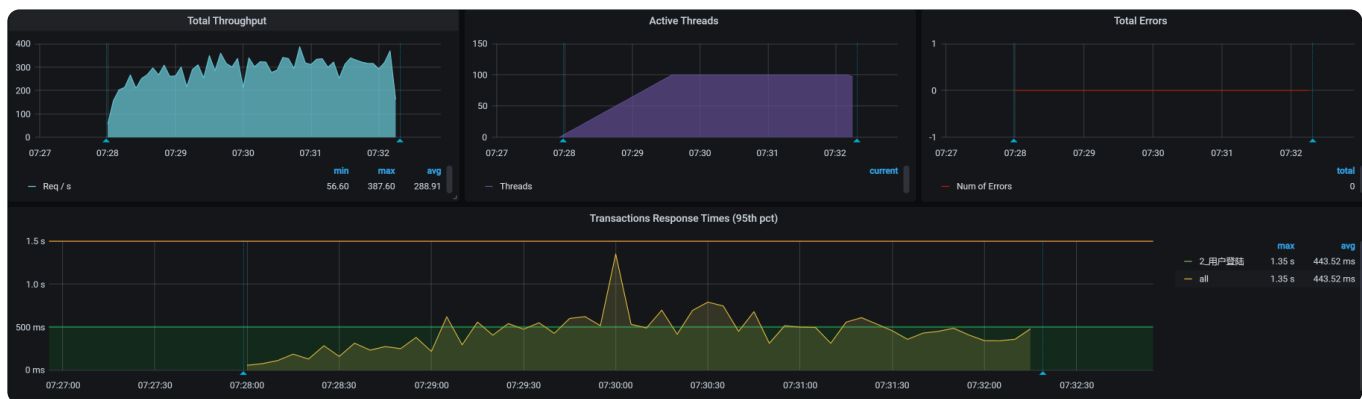
检验缓存的效果

为了确定缓存对后续的性能优化产生了效果，我们可以用两个手段来检验效果：

把参数化数据量降下来，只用少量的数据测试一下（请注意，我们只是尝试一下，并不是说用少量的数据来运行场景是对的）；

直接加载全部缓存。

我们得到这样的结果：



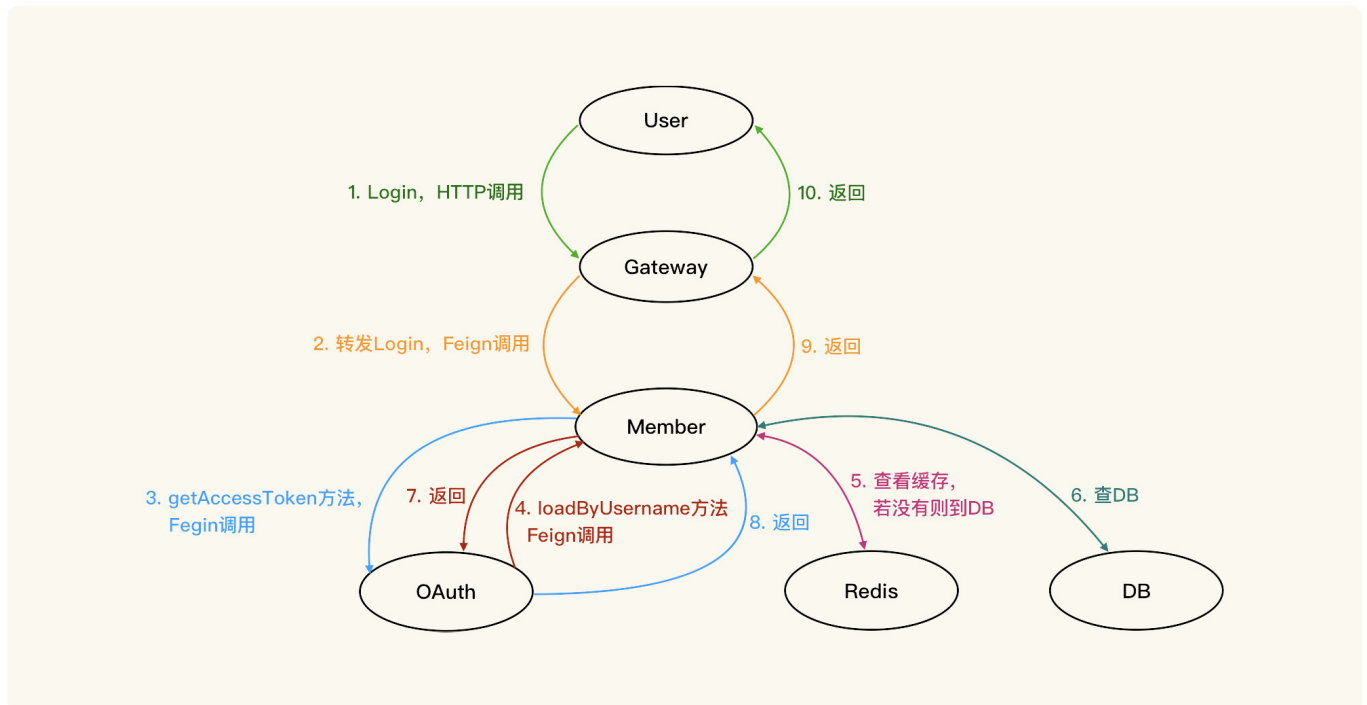
从曲线上看，登录接口能达到 300TPS 左右了。但是，我还是觉得不够好，因为从硬件资源上来看，再根据我以往的经验，它应该还能达到更高才对。

而在分析的过程中，再也没有第 11 讲和第 12 讲中提到的硬件资源的问题，但是在这里我们通过查看全局监控数据，看到的是 `us cpu` 高，说明确实都是业务逻辑在消耗

CPU 资源了。所以，我们就只有从登陆逻辑入手，来优化这个问题了。

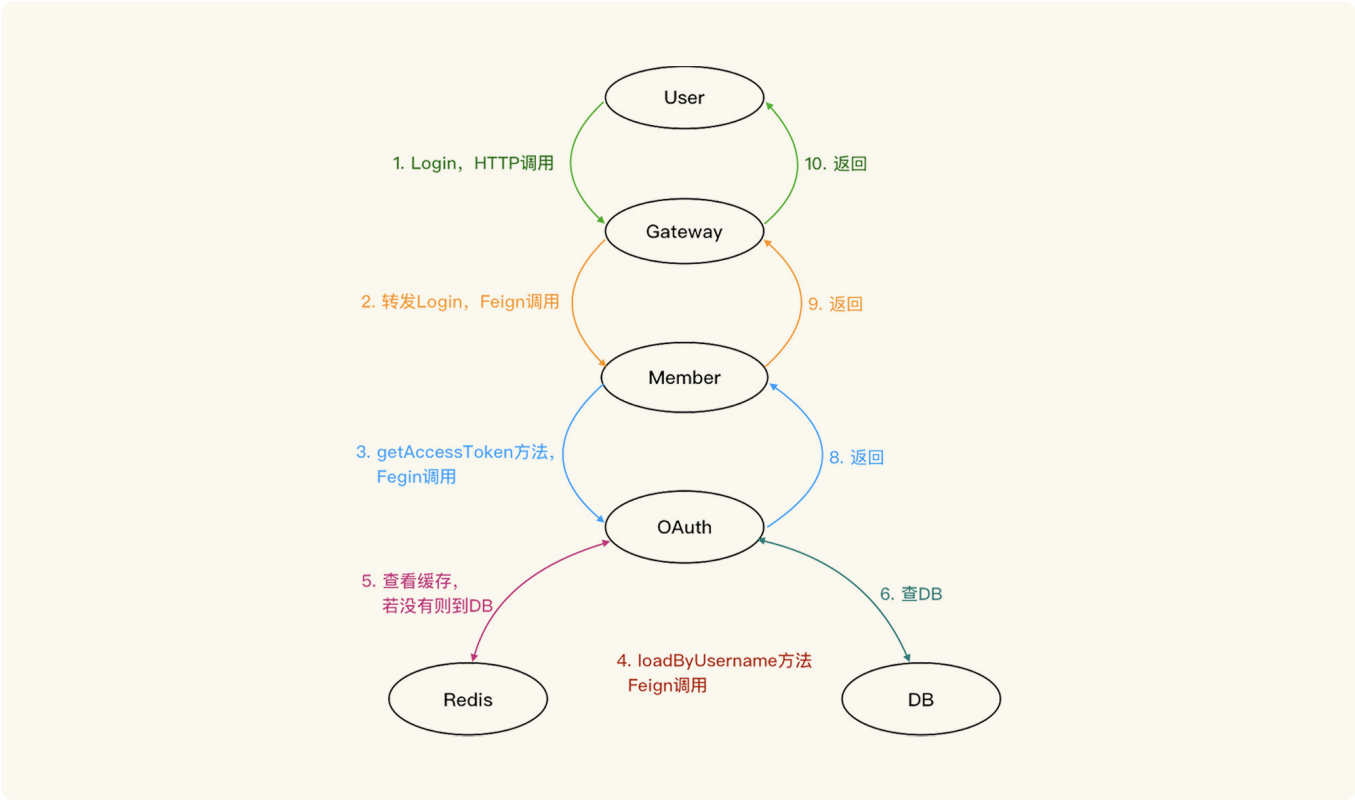
修改登录的逻辑

通过阅读源代码，我整理了这个系统的原登录逻辑：

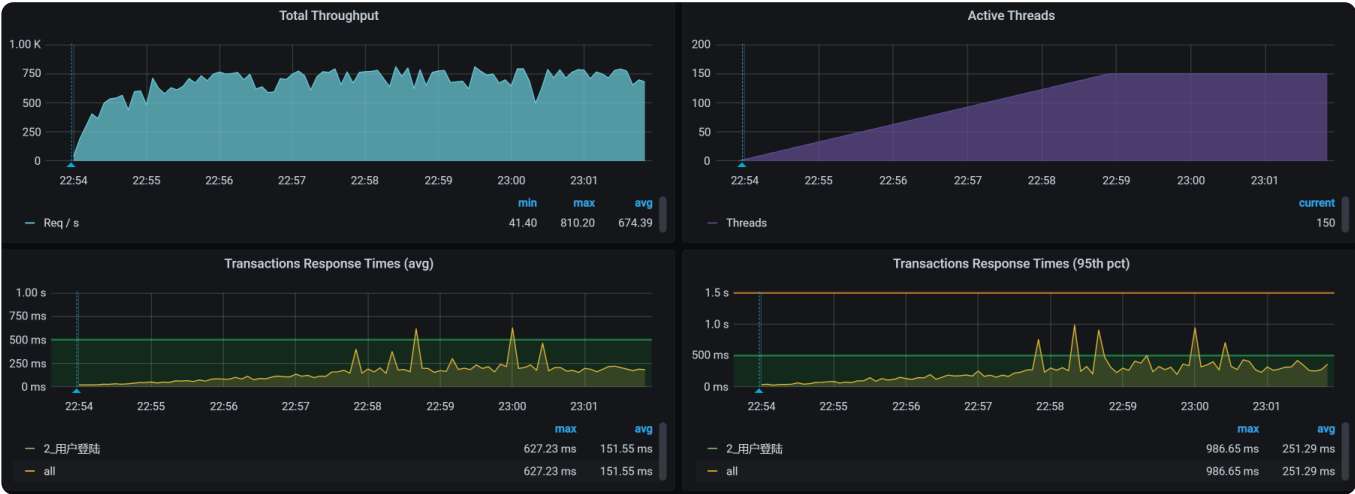


这个逻辑看着比较啰嗦，其中 Member 服务调 auth 服务，倒还能理解。可是，Auth 服务为什么还要到 Member 里取用户名呢？自己直接查缓存或 DB 不香吗？从架构设计的角度来看，为了避免共享数据库，这样的设计似乎也没啥。只是在我们的优化过程中，需要根据实际环境来做判断。

在我们这个环境中，需要把 DB 共用，这样 Auth 服务就可以直接使用数据库，而不用再从 Member 绕一圈。所以，我们先改成下面这种新的登录逻辑，这样就可以减少一次调用。



修改之后，登录 TPS 如下：

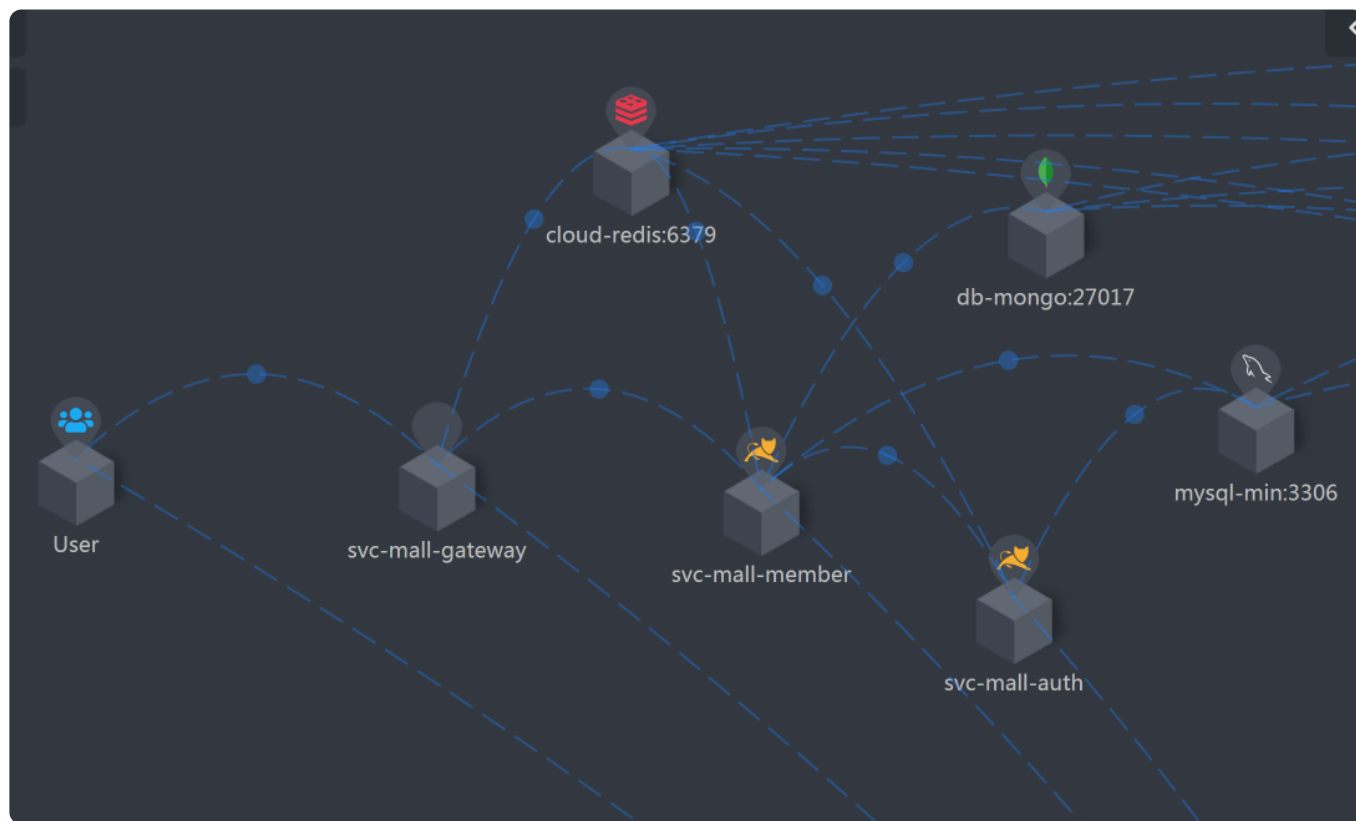


从结果上来看，TPS 确实有增加，已经到 700 以上了。很好。

这时候是不是就可以结束分析了呢？不是，我们还需要知道当前的瓶颈点在哪，因为根据我的性能理念，每个业务都会有瓶颈点，不管优化到什么程度，除非一直把硬件资源耗光。所以，我们继续接着分析。

看架构图

还是一样，在分析性能瓶颈之前，我们先来看架构图，了解用户登录接口会涉及到哪些服务和技术组件。

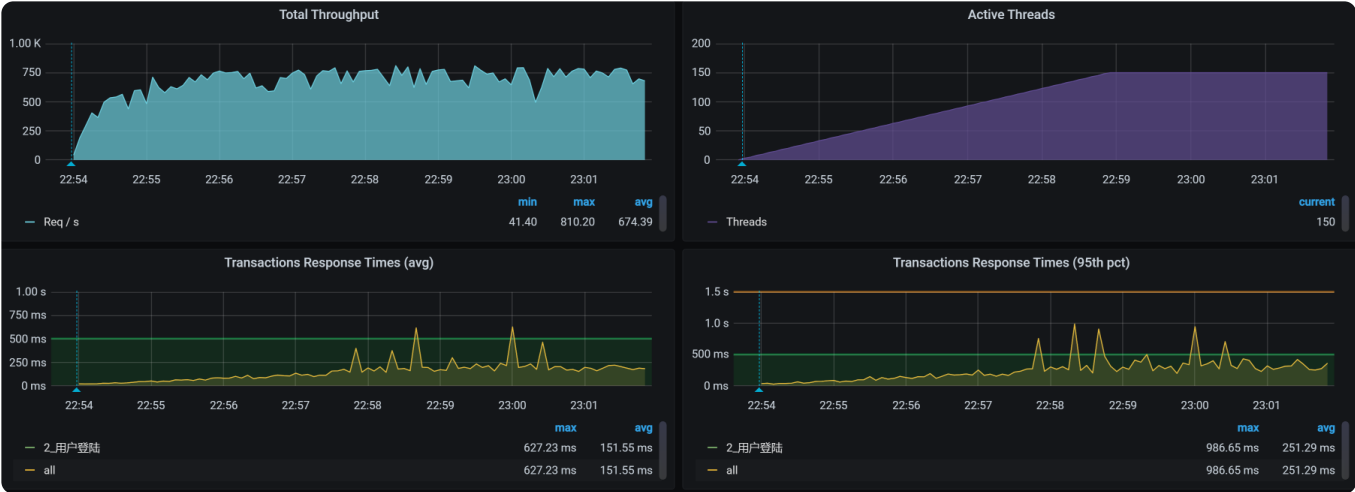


从这个架构图中可以看到，登录操作跨了 Gateway/Member/Auth 三个服务，连接了 Redis/MySQL 两个组件。图中的 MongoDB 虽然看上去有线，但实际上登录并没有用上。

了解这些信息之后，我们按照分析逻辑，一步步来分析问题。

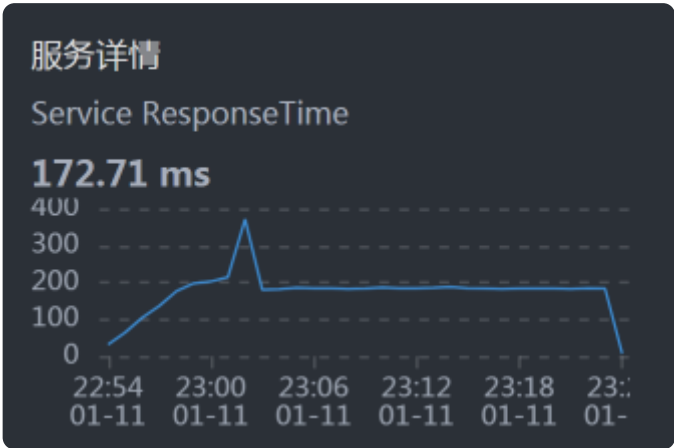
拆分时间

我们前面提到，修改登录逻辑后的 TPS 如下：



可以看到，响应时间已经上升到了 100ms 左右，所以，我们现在要找出这个时间消耗在了哪里。你可能已经注意到，图中的用户增加到了 150。这是为了把响应时间拉大，便于我们分析。下面我们把这个响应时间拆分一下，看看问题出在哪里。

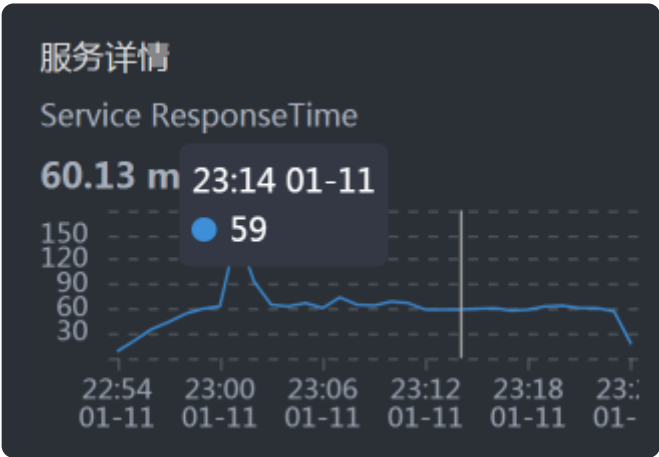
Gateway 服务上的时间



Member 服务上的时间



Auth 服务上的时间



我们看到，Member 服务上的时间消耗是 150ms 左右，Auth 服务上的时间消耗有 60ms 左右。Member 服务是我们着重分析的，因为它的响应时间更长。而 Auth 上虽然时间不太长，但是也达到了 60ms 左右，从经验上来说，我觉得还是有点稍长了，最好平均能到 50ms 以下，所以我们也要稍微关心一下。

全局监控

我们的分析逻辑雷打不动，依旧是**先看全局监控，后看定向监控**。从下面这张全局监控图的数据来看，worker-7 和 worker-8 的 CPU 使用率比其他的要高。

IP (链接到明细)	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	5.01 day	7.64 GiB	4	0.24	9.49%	21.46%	29.95%	0 B/s	100.93 kB/s	247	18	298.01 kbps	442.90 kbps
k8s-master-2	k8s-master-2	3.30 week	7.64 GiB	4	0.65	17.63%	23.07%	29.95%	0 B/s	77.12 kB/s	291	17	248.09 kbps	302.07 kbps
k8s-master-3	k8s-master-3	3.30 week	7.64 GiB	4	0.39	9.79%	26.13%	29.95%	0 B/s	77.42 kB/s	322	18	1.15 Mbps	1.15 Mbps
k8s-worker-1	k8s-worker-1	2.21 week	15.51 GiB	6	2.93	37.10%	75.87%	42.36%	37.76 kB/s	850.43 kB/s	28	94	60.21 Mbps	65.19 Mbps
k8s-worker-2	k8s-worker-2	2.21 week	15.51 GiB	6	6.39	58.26%	77.43%	29.98%	36.82 kB/s	218.61 kB/s	29	28 K	56.35 Mbps	61.96 Mbps
k8s-worker-3	k8s-worker-3	2.21 week	15.51 GiB	6	0.32	4.67%	31.99%	29.95%	9.21 kB/s	3.75 kB/s	28	134	64.72 kbps	82.99 kbps
k8s-worker-4	k8s-worker-4	2.21 week	15.51 GiB	6	0.60	13.21%	30.69%	31.65%	9.21 kB/s	10.74 kB/s	47	196	10.51 Mbps	11.38 Mbps
k8s-worker-5	k8s-worker-5	2.21 week	15.51 GiB	6	2.22	34.27%	43.69%	35.75%	13.81 kB/s	4.72 kB/s	28	104	18.31 Mbps	23.46 Mbps
k8s-worker-6	k8s-worker-6	2.21 week	15.51 GiB	6	0.71	17.31%	41.46%	38.88%	23.02 kB/s	5.39 kB/s	29	116	18.83 Mbps	19.80 Mbps
k8s-worker-7	k8s-worker-7	1.91 week	15.51 GiB	4	20.37	86.42%	28.79%	41.19%	23.02 kB/s	12.47 kB/s	32	708	15.76 Mbps	17.10 Mbps
k8s-worker-8	k8s-worker-8	5.14 day	15.51 GiB	4	17.98	84.91%	22.65%	43.76%	0 B/s	6.60 kB/s	29	749	27.63 Mbps	29.20 Mbps
k8s-worker-9	k8s-worker-9	1.91 week	15.51 GiB	6	0.15	4.28%	21.97%	29.95%	18.41 kB/s	3.48 kB/s	29	82	79.43 kbps	121.11 kbps

既然 worker-7/8 的 CPU 使用率要高一点，那我们就要查一下这两个节点上跑着什么样的服务。所以我们来看一下 POD 的分布，大概看一下每个 POD 在哪个 worker 节点上，以便后面分析 POD 相互之间的影响：

```
[root@k8s-master-1 ~]# kubectl get pods -o wide | grep mall
gateway-mall-gateway-869f84bd9-vbth4      1/1      Running    0          2d5h      10.100.140.42      k8s-worker-2      <none>      <none>
monitor-mall-monitor-d8bb58fcb-kfbcj      1/1      Running    0          29h       10.100.231.242     k8s-worker-8      <none>      <none>
svc-mall-admin-59ccc9ccc7-csv8j            1/1      Running    0          5d18h     10.100.231.199     k8s-worker-8      <none>      <none>
svc-mall-auth-576bd7bc4c-4pnr7             1/1      Running    0          5d18h     10.100.5.7         k8s-worker-7      <none>      <none>
svc-mall-cart-84c58f5c5b-f8mz7            1/1      Running    0          21h       10.100.5.63        k8s-worker-7      <none>      <none>
svc-mall-demo-5b66fbd9d5-8zwj7            1/1      Running    0          4d19h     10.100.231.202     k8s-worker-8      <none>      <none>
svc-mall-member-88457f776-vdgh9           1/1      Running    0          5d18h     10.100.231.197     k8s-worker-8      <none>      <none>
svc-mall-order-79d9b85bc4-czn25           1/1      Running    0          27h       10.100.5.58        k8s-worker-7      <none>      <none>
svc-mall-portal-7d8cf6ff49-nr5f6          1/1      Running    0          5d18h     10.100.69.195      k8s-worker-3      <none>      <none>
svc-mall-search-f6999bd98-182k7           1/1      Running    0          5d18h     10.100.5.5         k8s-worker-7      <none>      <none>
```

原来，在 worker-7 和 worker-8 上，分别运行着响应时间稍高的 Auth 服务和 Member 服务。对于这两个服务，我们都要分析，只是得一个一个来，那我们就从 auth 服务开始。

你可能会问：为什么要先从 Auth 服务下手呢？没啥原因，就是看它的 CPU 更红一点。你还可能奇怪：图中其他地方也红了，为什么不关注呢？我来逐一给你分析一下。

图中的 worker-1 和 worker-2，内存使用率相对较大，达到了 70% 以上。从经验上来说，我几乎没有怎么关心过 Linux 的内存使用率，除非出现大量的 page faults。因为 Linux 内存在分配给应用程序使用之后，是会体现在 Cache 当中的。被应用程序 Cache 住的内存在操作系统上来看都是被使用的，但实际上可能并未真的被使用，这时操作系统会把这部分 Cache 内存计算到 available 内存当中，所以说，我们直接看操作系统级别的空闲内存是分析不出问题来的。

在 worker-2 上，我们看到 TCP 的 Time Wait 达到近 3 万，不过这也不是我关心的点，因为 Time Wait 是正常的 TCP 状态，只有端口不够用、内存严重不足，我才会稍微看一眼。

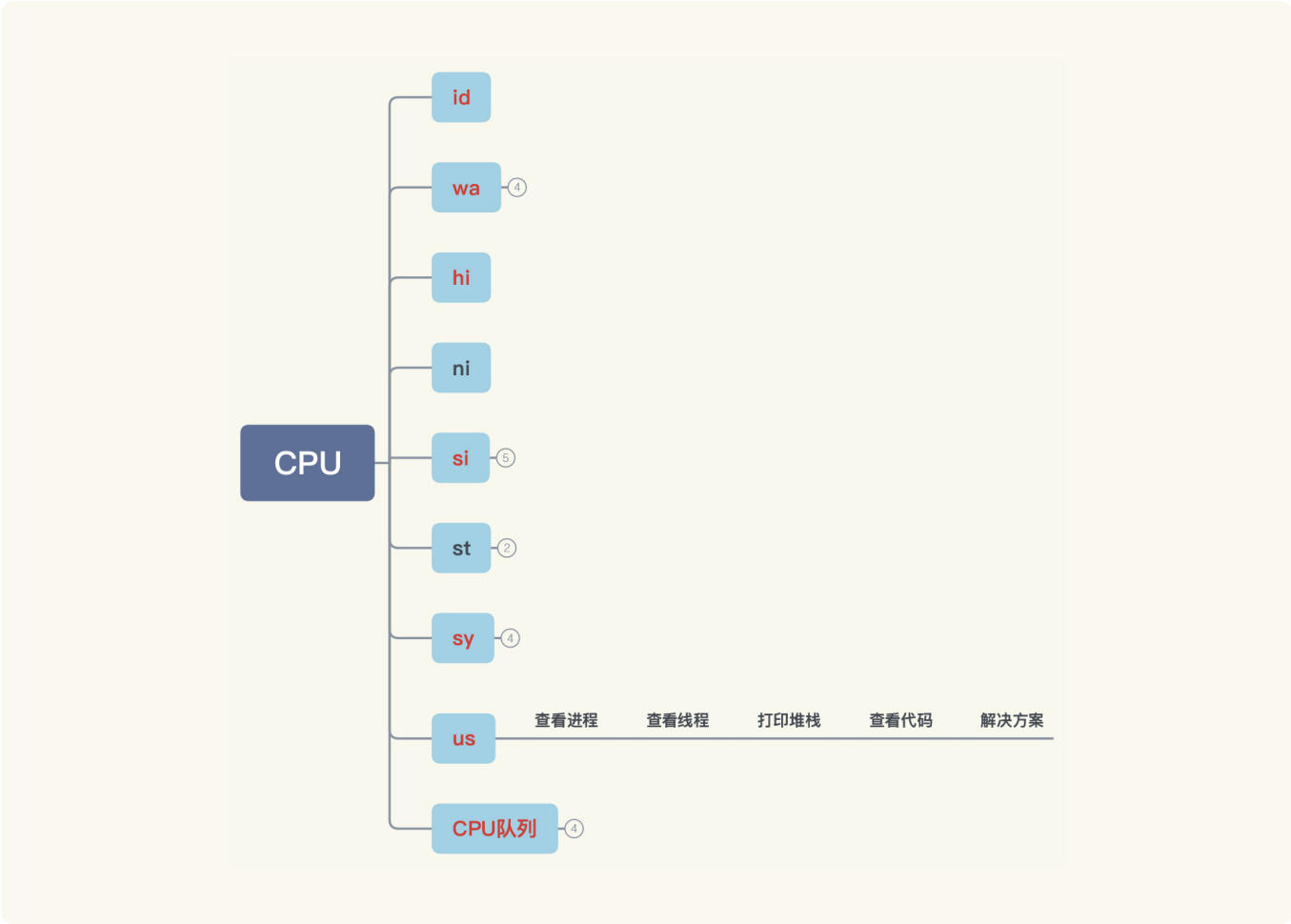
至于 worker-1 和 worker-2 的上下行带宽，看起来真是不大。在内网结构中，我们在测试的时候，内网带宽达到过好几 Gbps，这点带宽还不足以引起我们的重视。

所以，我们要“收拾”的还是 worker-7 和 worker-8。

既然 Auth 服务在 worker-7 上，member 服务在 worker-8 上，就像前面说的，我们不如就从 Auth 服务开始。

Auth 服务定向分析

对于 Auth 服务，我们从哪里开始分析呢？其实，我们可以按部就班。既然是 Auth 服务导致 worker-7 的 CPU 使用率偏高，那我们就可以走下面这个证据链：



按照这个证据链，我们应该先看进程。不过，仗着傻小子火气壮（俗称：艺高人胆大），我直接就去看线程状态了，想看看能不能凭经验蒙对一把。于是，我打开了 Spring Boot Admin 的线程页面：



有没有满目疮痍的感觉？人生就是这样，到处都有惊吓。

在我颤抖着手点开一些红色的地方之后，看到了类似这样的信息：

线程ID	86568
线程名称	http-nio-8401-exec-744
线程状态	BLOCKED
阻塞数	842
阻塞时间	-1
等待数	10515
等待时间	-1
锁名称	sun.security.rsa.RSACore\$BlindingParameters@34cfa67d
锁拥有者ID	86676
锁拥有者名称	http-nio-8401-exec-823

可以看到，阻塞数非常大，达到了 842。此外，锁拥有者 ID 是 86676，锁拥有者名称是线程 823。

我们抓两段栈出来看一下，找一下锁之间的关系：

 复制代码

```
1  -- 第一处：
2  "http-nio-8401-exec-884" #86813 daemon prio=5 os_prio=0 tid=0x00007f2868073000
3      java.lang.Thread.State: BLOCKED (on object monitor)
4      at java.security.Provider.getService(Provider.java:1035)
5      - waiting to lock <0x000000071ab1a5d8> (a sun.security.provider.Sun)
6      at sun.security.jca.ProviderList.getService(ProviderList.java:332)
7      .....
8      at com.dunshan.mall.auth.util.MD5Util.toMD5(MD5Util.java:11)
9      at com.dunshan.mall.auth.config.MyPasswordEncoder.matches(MyPasswordEncoder.
10     .....
11     at com.dunshan.mall.auth.controller.AuthController.postAccessToken$original$
12     at com.dunshan.mall.auth.controller.AuthController.postAccessToken$original$
13     at com.dunshan.mall.auth.controller.AuthController$auxiliary$z8kF9l34.call(U
14     .....
15     at com.dunshan.mall.auth.controller.AuthController.postAccessToken(AuthContr
16     .....
17
18
19  -- 第二处：
20  "http-nio-8401-exec-862" #86728 daemon prio=5 os_prio=0 tid=0x00007f28680d6000
21      java.lang.Thread.State: BLOCKED (on object monitor)
22      at sun.security.rsa.RSACore$BlindingParameters.getBlindingRandomPair(RSACore
```

```
23 - waiting to lock <0x0000000071ddad410> (a sun.security.rsa.RSACore$BlindingP
24 at sun.security.rsa.RSACore.getBlindingRandomPair(RSACore.java:443)
25 .....
26 at com.dunshan.mall.auth.controller.AuthController.postAccessToken$original$
27 at com.dunshan.mall.auth.controller.AuthController.postAccessToken$original$
28 at com.dunshan.mall.auth.controller.AuthController$auxiliary$z8kF9l34.call(U
29 .....
30 at com.dunshan.mall.auth.controller.AuthController.postAccessToken(AuthContr
31 .....
```

这两个栈的内容并不是同一时刻出现的，说明这个 BLOCKED 一直存在。但是不管怎么样，这个栈在做 RSA 加密，它和 Token 部分有关。

其中，线程 http-nio-8401-exec-884 是 BLOCKED 状态，那就说明有其他线程持有这个锁，所以我们自然要看一下线程栈中的 waiting to lock <0x0000000071ab1a5d8>。其实，如果你有经验的话，一下子就能知道这里面是什么问题。不过，我们做性能分析的人要讲逻辑。

我在这里啰嗦几句，**当你碰到这种锁问题，又不知道具体原因的时候，要下意识地去打印一个完整的栈来看，而不是再到 Spring Boot Admin 里胡乱点。**为什么不建议你这么做？原因有这么几个：

由于线程太多，点着看逻辑关系比较累；

不断在刷，眼晕；

我不喜欢。

所以，对于前面遇到的锁问题，我们首先要做的就是到容器中的 jstack 里打印一下栈，把它下载下来，然后祭出工具打开看一眼。

你可能会问，为什么不用 Arthas 之类的工具直接在容器里看？主要是因为 Arthas 的 Dashboard 在 Thread 比较多的时候，看起来真心累。

下面这张图就是 jstack 打印出来的栈，在下载之后用工具打开的效果。

Name	Timestamp	Runnable/Total Threads
auththreaddump.txt_1	Jan 11 23:51:39 2021	15/224

- File name : /Users/gaolou/Desktop/Zee的软件性能工程/0-软件性能工程/1_性能测试实践专栏/auth阻塞分析/auththreaddump.txt
- Timestamp : 2021-01-11 23:51:39
- Thread Status Analysis

Status	Number of Threads : 224	Percentage
 Deadlock	0	0 (%)
 Runnable	15	7 (%)
 Waiting on condition	201	90 (%)
 Waiting on monitor	0	0 (%)
 Suspended	0	0 (%)
 Object.wait()	4	2 (%)
 Blocked	4	2 (%)
 Parked	0	0 (%)

是不是有种买彩票的感觉？看起来有那么多多的 BLOCKED 状态的线程（多达 842 个），居然一个都没蒙到！我本来想抓 BLOCKED 状态的线程，并且线程描述是 “Waiting on monitor”，但是，从上面的线程描述统计来看，一个也没见。哼，真生气。

这时候，身为一个做性能分析的人，我们一定要记得倒杯茶，静静心，默默地把 jstack 连续再执行几遍。我在这里就连续执行了 10 遍，然后再找每个栈的状态。

终于，Waiting on monitor 来了：

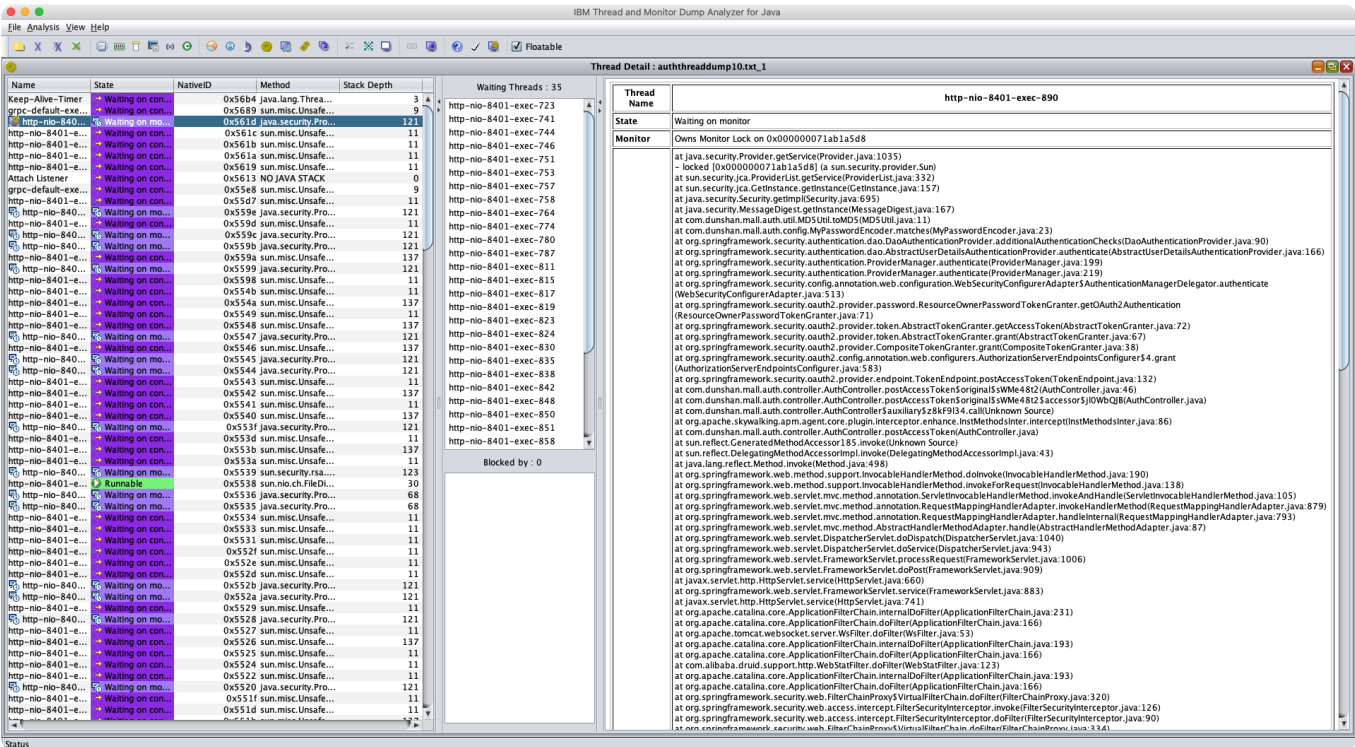
Status	Number of Threads : 229	Percentage
 Deadlock	0	0 (%)
 Runnable	20	9 (%)
 Waiting on condition	153	67 (%)
 Waiting on monitor	47	21 (%)
 Suspended	0	0 (%)
 Object.wait()	4	2 (%)
 Blocked	5	2 (%)
 Parked	0	0 (%)

Status	Number of Threads : 228	Percentage
Deadlock	0	0 (%)
Runnable	32	14 (%)
Waiting on condition	157	69 (%)
Waiting on monitor	31	14 (%)
Suspended	0	0 (%)
Object.wait()	4	2 (%)
Blocked	4	2 (%)
Parked	0	0 (%)

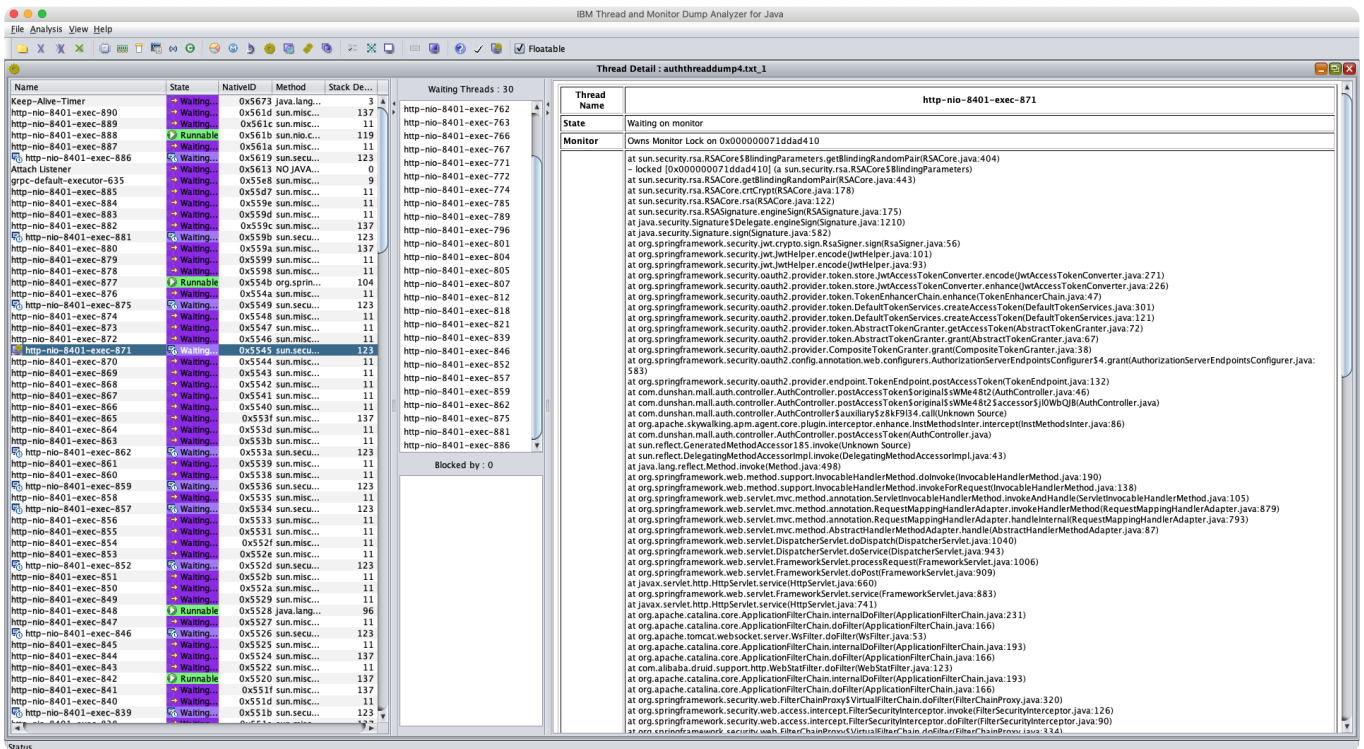
看起来有得玩了！接下来让我们看看究竟是谁阻塞住了上面的线程。

我们先在相应的栈里，找到对应的持有锁的栈。下面是栈中的阻塞关系。

第一个栈



第二个栈



你要注意，这是两个栈文件。所以，我们要分别从这两个栈文件里找到各自的对应等待关系。下面这段代码就对应了上面的 Waiting 线程。

[复制代码](#)

```
1  -- 第一处
2  "http-nio-8401-exec-890" #86930 daemon prio=5 os_prio=0 tid=0x000007f28680a5800
3      java.lang.Thread.State: BLOCKED (on object monitor)
4      at java.security.Provider.getService(Provider.java:1035)
5      - locked <0x000000071ab1a5d8> (a sun.security.provider.Sun)
6      at sun.security.jca.ProviderList.getService(ProviderList.java:332)
7      .....
8      at com.dunshan.mall.auth.util.MD5Util.toMD5(MD5Util.java:11)
9      at com.dunshan.mall.auth.config.MyPasswordEncoder.matches(MyPasswordEncoder.
10     .....
11     at com.dunshan.mall.auth.controller.AuthController.postAccessToken$original$
12     at com.dunshan.mall.auth.controller.AuthController.postAccessToken$original$
13     at com.dunshan.mall.auth.controller.AuthController$auxiliary$z8kF9l34.call(U
14     at org.apache.skywalking.apm.agent.core.plugin.interceptor.enhance.InstMetho
15     at com.dunshan.mall.auth.controller.AuthController.postAccessToken(AuthContr
16
17
18  -- 第二处
19  "http-nio-8401-exec-871" #86739 daemon prio=5 os_prio=0 tid=0x000007f28681d6800
20      java.lang.Thread.State: BLOCKED (on object monitor)
21      at sun.security.rsa.RSACore$BlindingParameters.getBlindingRandomPair(RSACore
22      - locked <0x000000071ddad410> (a sun.security.rsa.RSACore$BlindingParameters
23      at sun.security.rsa.RSACore.getBlindingRandomPair(RSACore.java:443)
24      .....
25      at com.dunshan.mall.auth.controller.AuthController.postAccessToken$original$
26      at com.dunshan.mall.auth.controller.AuthController.postAccessToken$original$
```

```
27 at com.dunshan.mall.auth.controller.AuthController$auxiliary$z8kF9l34.call(U
28 at org.apache.skywalking.apm.agent.core.plugin.interceptor.enhance.InstMetho
29 at com.dunshan.mall.auth.controller.AuthController.postAccessToken(AuthContr
```

你看上面 locked 这一行的锁 ID，既然找到了这两处持有锁的栈，那我们就通过上面的栈，到源码中找到这两处栈的代码：

 复制代码

```
1  -- 第一处同步代码块
2      public synchronized Service getService(String type, String algorithm) {
3          checkInitialized()
4          // avoid allocating a new key object if possible
5          ServiceKey key = previousKey
6          if (key.matches(type, algorithm) == false) {
7              key = new ServiceKey(type, algorithm, false);
8              previousKey = key
9
10         if (serviceMap != null) {
11             Service service = serviceMap.get(key)
12             if (service != null) {
13                 return service;
14
15
16             ensureLegacyParsed()
17             return (legacyMap != null) ? legacyMap.get(key) : null;
18
19
20
21  -- 第二处同步代码块
22      // return null if need to reset the parameters
23      BlindingRandomPair getBlindingRandomPair(
24          BigInteger e, BigInteger d, BigInteger n) {
25
26
27          if ((this.e != null && this.e.equals(e)) ||
28              (this.d != null && this.d.equals(d))) {
29
30
31              BlindingRandomPair brp = null;
32              synchronized (this) {
33                  if (!u.equals(BigInteger.ZERO) &&
34                      !v.equals(BigInteger.ZERO))
35
36
37                      brp = new BlindingRandomPair(u, v);
38                  if (u.compareTo(BigInteger.ONE) <= 0 ||
39                      v.compareTo(BigInteger.ONE) <= 0) {
40
```

```
41         // need to reset the random pair next time
42         u = BigInteger.ZERO
43         v = BigInteger.ZERO
44     } else {
45         u = u.modPow(BIG_TWO, n)
46         v = v.modPow(BIG_TWO, n)
47
48     } // Otherwise, need to reset the random pair.
49
50     return brp;
51
52
53
54     return null;
55
```

你可以看到，第一处是 JDK 中提供的 `getService` 类采用了全局同步锁定，导致的分配 key 时产生争用，这个其实在 JDK 的 Bug List 中有过描述，详见 [JDK-7092821](#)。准确来说，它不算是 Bug，如果你想改的话，可以换一个库。

第二处是 JDK 中提供的 RSA 方法，是为了防范时序攻击特意设计成这样的。RSA 中有大素数的计算，为了线程安全，RSA 又加了锁。关于 RSA 的逻辑，你可以去看下源代码的 `/sun/security/rsa/RSACore.java` 中的逻辑。

不过，RSA 是一种低效的加密方法，当压力发起来的时候，这样的 `synchronized` 类必然会导致 BLOCKED 出现。对此，在源码中有下面这样一段注释，其中建议先计算 `u/v`，可以提高加密效率。

[复制代码](#)

```
1  * Computing inverses mod n and random number generation is slow, s
2  * it is often not practical to generate a new random (u, v) pair for
3  * each new exponentiation. The calculation of parameters might even be
4  * subject to timing attacks. However, (u, v) pairs should not be
5  * reused since they themselves might be compromised by timing attacks,
6  * leaving the private exponent vulnerable. An efficient solution to
7  * this problem is update u and v before each modular exponentiation
8  * step by computing:
9  *
10 *     u = u ^ 2
11 *     v = v ^ 2
12
13 * The total performance cost is small
```

既然我们已经知道了这两个 BLOCKED 产生的原因，那下一步的操作就比较简单了。

针对第一处锁：实现自己的方法，比如说实现一个自己的分布式锁。

针对第二处锁：换一个高效的实现。

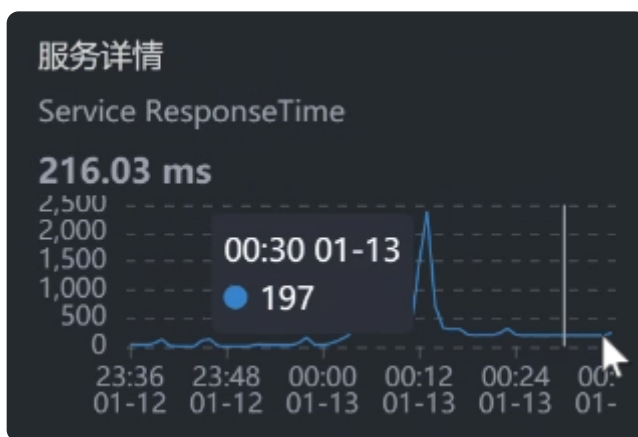
至此，我们就找到了应用中 BLOCKED 的逻辑。因为我们这是一个性能专栏，所以我不再接着整下去了。如果你是在一个项目中，分析到这里就可以把问题扔给开发，然后去喝茶了，让他们伤脑筋去，哈哈。

不过，这只是一句玩笑而已，你可别当真。作为性能分析人员，我们要给出合情合理并且有证据链的分析过程，这样我们和其他团队成员沟通的时候，才会更加简单、高效。

Member 服务定向分析

分析完 Auth 服务后，我们再来看看 Member 服务的性能怎么样。因为全局监控数据前面我们已经展示了，所以这里不再重复说明，我们直接来拆分一下对 Member 服务调用时的响应时间。

Gateway 上的响应时间



Member 上的响应时间



Auth 上的响应时间



从上面的信息来看，这几段都有不同的时间消耗：Member 服务上有 80 毫秒左右，Auth 服务上已经有 60 毫秒左右，明显是有点高了。

我们登录到 Member 这个服务中，先看整体的资源使用情况。这里我用了最经典传统的 top 命令：

复制代码

```
1 %Cpu0 : 63.8 us, 12.4 sy, 0.0 ni, 9.2 id, 0.0 wa, 0.0 hi, 14.2 si, 0.4 s
2 %Cpu1 : 60.3 us, 11.7 sy, 0.0 ni, 11.0 id, 0.0 wa, 0.0 hi, 16.6 si, 0.3 s
3 %Cpu2 : 59.4 us, 12.0 sy, 0.0 ni, 14.1 id, 0.0 wa, 0.0 hi, 13.8 si, 0.7 s
4 %Cpu3 : 59.8 us, 12.1 sy, 0.0 ni, 11.7 id, 0.0 wa, 0.0 hi, 15.7 si, 0.7 s
```

从 CPU 使用分布上来看，其他计数器都还正常，只是 si 有点高。这是一个网络中断的问题，虽然有优化的空间，但是受基础架构所限，性能提升得不太多，这也是为什么现在很多企业都放弃了虚拟化，直接选择容器化的一个原因。

针对这个网络中断的问题，我将在后面的课程中仔细给你扒一扒，这节课我们暂且不做过多的讲解。

总结

这节课我用登录功能给你串了一个完整的性能分析场景。

在前面代码修改的部分，性能分析过程是比较快的，我们就是看看哪里的代码逻辑会消耗更多的时间。这个思路就是前面提到的 us cpu 的证据链。

而接下来我们在分析 Auth 服务的时候，是先从拆分时间开始一步步走到代码里的，其中最核心的部分是从 CPU 到栈，再到 BLOCKED 的判断。当我们看到栈上有 BLOCKED 的时候，要记得打印栈信息。但是因为有些锁会非常快速地获取和释放，所以就可能会出现打印栈时，看到等某个锁的栈信息，但是整个栈文件中却没有这把锁的情况。这个时候，你就要注意了，**我们一定要去连续地多打几次栈，直到抓到对应的锁。**

这是分析栈中锁的一个关键，因为我们经常会看到等锁的栈信息，看不到持有锁的栈信息。而连续多打几次栈，就是为了把持有锁和等待锁的栈同时打印出来，否则我们就找不出分析的逻辑了。

接着，当我们看到了持有锁的栈之后，就根据自己业务代码的调用逻辑，一层层地去找是哪里加的锁。至于这个锁加的合理不合理，就和业务逻辑有关了。作为性能分析人员，这个时候我们就可以把开发、业务、架构等人拉到一起讨论。这个锁要不要改，不是做性能的人说了算，而是大家一起说了算。

通过上述的分析，相信你可以看到，在我的性能分析逻辑中，从现象到原理，都需要搞清楚。

课后作业

最后，我给你留几个思考题来巩固今日所学。

1. 为什么看到 BLOCKED 的栈时要连续多打几次栈信息？
2. 为什么从性能分析中要从现象到原理？
3. 低效的代码有什么优化思路？

记得在留言区和我讨论、交流你的想法，每一次思考都会让你更进一步。

如果你读完这篇文章有所收获，也欢迎你分享给你的朋友，共同学习进步。我们下一讲再见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 12 | 打开首页之二：如何平衡利用硬件资源？

下一篇 14 | 用户信息查询：如何解决网络软中断瓶颈问题？

精选留言 (2)

写留言



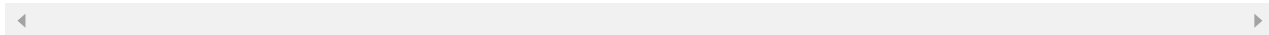
继贤

2021-04-19

老师，堆栈分析工具用的是哪个？

展开 ∨

作者回复: 我这里用的是IBM Thread and Monitor Dump Analyzer for Java。你也可以用其他工具，直接用文本看也是一样的。



1



浅浅

2021-04-25

我们抓两段栈出来看一下，找一下锁之间的关系：

老师，这个是用什么抓取的呀？

展开 ∨

作者回复: 这个用jstack、jvisualvm、spring boot admin、arthas，都可以抓呀。这个在前面的工具中已经罗列了哇。

